UMIACS-TR-96-31.1 October, 1996
CS-TR-3636.1

Exploiting Monotone Convergence Functions in Parallel Programs

William Pugh
pugh@cs.umd.edu
Inst. for Advanced Computer Studies
Dept. of Computer Science

Evan Rosser
ejr@cs.umd.edu
Dept. of Computer Science

Tatiana Shpeisman
murka@cs.umd.edu
Dept. of Computer Science

Univ. of Maryland, College Park, MD 20742

Abstract

Scientific codes which use iterative methods are often difficult to parallelize well. Such codes usually contain `while` loops which iterate until they converge upon the solution. Problems arise since the number of iterations cannot be determined at compile time, and tests for termination usually require a global reduction and an associated barrier. We present a method which allows us avoid performing global barriers and exploit pipelined parallelism when processors can detect non-convergence from local information.

This work is supported by an NSF PYI grant CCR-9157384 and by a Packard Fellowship.

Exploiting Monotone Convergence Functions in Parallel Programs

William Pugh, Evan Rosser, and Tatiana Shpeisman

Department of Computer Science
University of Maryland
{pugh,ejr,murka}@cs.umd.edu

Abstract. Scientific codes which use iterative methods are often difficult to parallelize well. Such codes usually contain `while` loops which iterate until they converge upon the solution. Problems arise since the number of iterations cannot be determined at compile time, and tests for termination usually require a global reduction and an associated barrier. We present a method which allows us avoid performing global barriers and exploit pipelined parallelism when processors can detect non-convergence from local information.

1 Introduction

Many scientific programs solve problems iteratively; that is, they compute an approximation to a solution, check if the approximation is sufficiently accurate (check for convergence), and conditionally perform another iteration.

In most instances, the loops inside the `while` are parallel, with data only needing to be communicated from one iteration to the next. However, the convergence test requires a global reduction and barrier, which can impose substantial performance penalties on some systems.

In a few cases, such as when a natural ordering is used in a relaxation algorithm, the inner loops carry dependencies and cannot be run in parallel. To exploit parallelism in these loops, a number of researchers [1,4,5,2,7,3] have proposed *speculative execution*: a wavefront technique is used to execute the program in parallel, despite the fact that all loops carry dependences. Since this ignores the termination condition of the `while` loop, iterations of the `while` loop are executed speculatively until each iteration is completely executed and it can be determined that the loop will continue past that iteration.

For both of these situations, we propose that we recognize and exploit a common pattern: that the convergence condition depends monotonically on looking at more and more data: if, from looking at a subset of the data we can determine that the while loop has not terminated, looking at more data will not change that decision.

In particular, each processor can check to see it can determine that a while loop continues just from looking at local data. If so, it can start on the next iteration without waiting for the global reduction to complete. Figure 1 shows the advantages conferred by eliminating this dependency. In programs where the body of the while loop can be executed in parallel, this allows us to avoid the penalties imposed by a global barrier. In the case where the body of the while loop contains dependences, this can often allow us to obtain doacross/pipelined parallelism.

If this idea is to be exploited, it is important that it be provided or supported by the compiler. Unless the program is written in explicitly parallel form, there is no way for a user to write a program that computes a reduction on just all local data, and then goes on to compute a global reduction if needed.

	No cross-processor dependences within an iteration of the while loop	Cross-processor dependences within an iteration of the while loop
Waiting for global reduction		
Able to continue without waiting for global reduction		

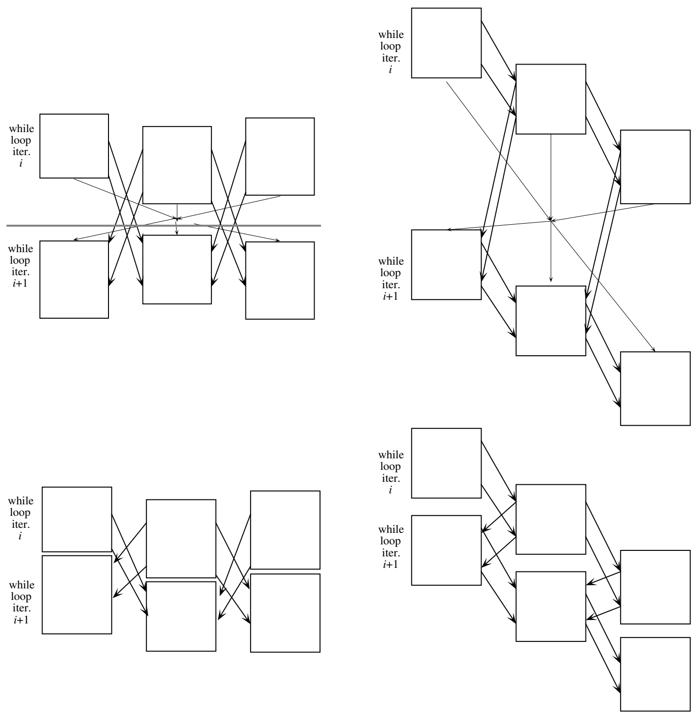

Fig. 1. Advantages of removing dependency on global reduction

In this paper, we discuss:

- What is required to recognize such patterns
- What code must be generated to exploit such patterns
- Experimental studies for the benchmarks SOR with Chebyshev acceleration and `tomcatv`.

2 Exploiting local information

In checking for convergence, the costs involved are the wait incurred by all processors while the result is computed, and that of the communication and synchronization associated with the reduction. To avoid these costs, we would like to do as much computation as possible on the local processor, and we would like to determine as quickly as possible the outcome of the convergence check. Specifically, if any processor could determine locally that the computation would not converge at this iteration, it could continue execution without fear that results could not be used. In this section, we first describe how to detect opportunities for this optimization; then we describe how we transform the program to take advantage of these properties.

2.1 Detecting monotone convergence functions

There are two aspects to detecting when our optimization can be applied. First, we need to detect the fact that the program has the loop structure that makes our optimization possible. Second, we need to determine if the function that checks for convergence has the monotonicity property we require.

Since it is a well-studied problem, we assume that any while loops that exist implicitly (with `if` statements and `goto`s) have already been recognized. Recognizing the pattern is then straightforward. We look for a pattern in which there is an outermost `while` loop, followed by a nest of `for` loops, and finally containing a global reduction, with a scalar data dependence to the `while` loop test.

The second aspect is detecting that the condition checking `while` loop termination can be computed locally with only a portion of the data. In other words, adding more data from other processors will not change the result of the function, and the result of the condition will change only once, from false to true. We wish to detect a number of common cases that can be recognized without extensive analysis. One condition which meets this criterion is checking if $x \geq y$, where x is the result of a reduction that is non-decreasing as more data is added, and y is a value that can be computed locally and is identical on all processors. In iterative codes, this pattern exists when checking to see if the current solution exceeds the acceptable error. Analogously, $x \leq y$ works when x is the result of a non-increasing function. A conjunction or disjunction of such conditions is also acceptable, as is a conjunction with a condition on the maximum number of iterations to perform (or any other scalar condition that can be computed on each processor from local data). Since this pattern is common in iterative codes, recognizing this pattern is sufficient for a number of programs.

Now we must characterize functions which are non-decreasing or non-increasing:

- Sum of non-negative numbers (as from absolute value or square)
- Maximum reductions

Non-increasing:

– Minimum reductions

If x_1, x_2 are results of non-decreasing (non-increasing) functions, the following are non-decreasing (non-increasing):

– $\sqrt{x_1}$, if x_1 is known to be positive
– $x_1 * x_2$, if x_1 and x_2 both positive (negative)
– $x_1 * y$ or x_1/y, where y is non-negative and invariant in the while loop

As an example, here are three commonly-used non-decreasing norms that fall into this category:

– $||x||_\infty$ (infinity norm) : $max_{i=1}^n |x_i|$
– $||x||_2$ (second norm) : $\sqrt{\sum_{i=1}^n x_i^2}$
– $||x||_1$ (first norm) : $\sum_{i=1}^n |x_i|$

It is feasible to detect these patterns in many real codes. In more complicated codes, a user directive might be useful to inform the compiler that the optimization is possible.

2.2 Changes to the code

In this section, we describe how the modified program will proceed on each processor.

New variables In order to take advantage of partial information about convergence, we need to keep track of several quantities in addition to the original program variables. These variables fall into two categories: first, those that record progress on the each processor, and those that record progress across all processors. One processor is designated as the master processor, which will handle the global reduction.

Each processor must keep track of how many iterations w of the while loop it has executed. This is necessary in order to provide the basis for processors to compare their relative progress through the program.

The master processor handles the remaining variables. First, it must record information about the global progress. If it is known that iteration w of the while loop will be executed, then it follows that all iterations $w' < w$ will also be executed. Therefore, the designated processor only need to know the number of w_{max}, the last iteration that is known not to converge. Each local processor keeps a local copy of this variable, $local\_w_{max}$.

The master processor must also combine partial reduction results. For iteration w_{max}, some processors may have completed their portion of the computation, but found that their portion of the reduction was not enough to prove non-convergence. The master processor accumulates results from each processor that has completed iteration w_{max}, to determine if a combination of individual contributions can prove non-convergence. In addition, on the last iteration, the accumulation represents the result of the global reduction once all processors have finished.

```
for(n = 1; n<= MAXITS; n++)
  rnorm = 0.0;
  jsw = 1;
  for(ipass=1; ipass<=2; ipass++)
    lsw = jsw;
    for(j = 2; j < jmax; j++)
      for(l = lsw+1; l<jmax; l+=2)
        resid=...
        rnorm += fabs(resid);
        u[j][l] -= omega*resid/-4;
      lsw=3-lsw;
    jsw = 3-jsw;
    omega=...;
  if(rnorm < EPS) return;
```

```
for(n = 1; n<= MAXITS; n++)
  local_rnorm = 0.0;
  jsw = 1;
  for(ipass=1; ipass<=2; ipass++)
    lsw = jsw;
    for(j = 2; j < jmax; j++)
      for(l = local_min+lsw+1;
          l<local_max;
          l+=2)
        resid=...
        local_rnorm += fabs(resid);
        u[j][l] -= omega*resid/-4;
      lsw=3-lsw;
    jsw = 3-jsw;
    omega=...;
  // Check termination locally.
  w = n;
  if(! w < local_w_max)
    send(master_proc, w, local_rnorm)
    if (! local_rnorm < EPS)
      // Can't proceed on local
      // information
      receive(master_proc, new_w);
      if (new_w == TERMINATE)
        return;
      else
        local_w_max = new_w;
```

Fig. 2. Pseudo-code for SOR with Chebyshev acceleration, before and after transformation

Checking non-convergence In this section, we describe how an individual processor decides whether it can safely proceed to the next iteration without waiting for the result of the global reduction.

Each processor p can proceed without waiting if $local\_w_{max} > w_p + 1$; that is, another processor has already detected non-convergence at iteration w, and that information was sent to p in a previous iteration.

Otherwise, the processor performs its local portion of the reduction. If it indicates non-convergence, it sends a pair of values $(w, local\_reduction)$ to the master processor, and continues to the next `while` iteration.

If the local reduction does not allow p to continue, it sends the values as above, and waits for a reply from the master processor. The reply will be either an iteration number, indicating that it is safe to go on, or a message that the program should terminate. If the response indicates non-convergence, the processor p saves the iteration number into $local\_w_{max}$, allowing it to avoid checking again until $w_p = local\_w_{max}$.

The master processor operates as follows. Upon receiving a pair of values $(w_p, reduction_p)$ from a processor p, it checks the global progress as indicated by w_{max}. If $w_p < w_{max}$, this message does not help prove more progress through the program. If $reduction_p$ shows that p was not able to prove non-convergence by itself, p must be waiting for a reply, and w_{max} is returned; $reduction_p$ is discarded as unnecessary. If $w_p = w_{max}$, then the $reduction_p$ portion is added into the growing global reduction. If the most recent contribution is enough to detect non-convergence, then the master processor sets w_{max} to $w_p + 1$, and sends it to all waiting processors to indicate that they can proceed (including p if $reduction_p$ was not sufficient by itself to prove this fact). Finally, if the master processor still cannot tell that the loop will continue after w_{max}, it adds p to the list of waiting processors and waits for more data.

If p is the last processor to report, and the master processor finds that the computation has converged, the master processor sends a message indicating termination to all processors. The partial reduction now contains the value for the global reduction.

An improvement to this scheme is to delay performing reduction communication between the worker processors and the master processor until at least one processor needs help in proving non-convergence. At that point, the processor in question sends a request to the master, which instructs all other processors to begin sending reduction messages. At each iteration, a processor probes to see if such a message from the master processor has arrived.

Communication between individual processors is not affected by this optimization.

An example of the transformed code appears in Figure 2. The code is adapted from [8].

3 Example: SOR

In this section we present an example program that can be parallelized using our method.

The major program class arises from solving partial differential equation (PDE) boundary value problems using finite-difference methods [8,6]. For example, consider solving a partial differential equation of the following form:

$$
\begin{aligned}
&f_1(x,y) \cdot \partial u^2/\partial x^2 + f_2(x,y) \cdot \partial u^2/\partial y^2 + f_3(x,y) \cdot \partial u/\partial x + \\
&f_4(x,y) \cdot \partial u/\partial y + f_5(x,y) \cdot u(x,y) = f_0(x,y) \qquad (1)
\end{aligned}
$$

Given the open region Ω in $\mathcal{R}^2$ and a function $g(x,y)$, the problem is to find such a function u that is continuous on the closure of Ω, satisfies Equation 1 in Ω, and equals g on the boundary.

Discretizing this problem on the $N \times N$ mesh using finite-difference method leads to the following discrete problem:

$$
\begin{aligned}
&a_{j,l} \cdot u_{j+1,l} + b_{j,l} \cdot u_{j-1,l} + c_{j,l} \cdot u_{j,l+1} + d_{j,l} \cdot u_{j,l-1} + e_{j,l} \cdot u_{j,l} + f_{j,l} = 0,\\
&\quad \text{for } j = 2, N-1 \text{ and } l = 2, N-1\\
&u_{j,l} = g_{j,l},\\
&\quad \text{for } j = 1 \text{ or } j = N \text{ or } l = 1 \text{ or } l = N
\end{aligned}
\tag{2}
$$

As a particular example of PDE boundary value problem we shall consider solving Laplace's equation $\partial u^2/\partial x^2 + \partial u^2/\partial y^2 = 0$ on the region $\Omega = \{(x, y) : 0 < x < 1, 0 < y < 1\}$ with the Dirichlet boundary conditions defined by function $g(x, y) = \sinh(3\pi y) \cdot \sinh(3\pi y) * 10^{-3}$ [6].

```
// Set the initial guess for u_{j,l}
for l = 1 to N do
    for j = 1 to N do
        if j = 1 or j = N or l = 1 or l = N
            then
                u_{j,l} = g((j-1)/N, (l-1)/N)
            else
                u_{j,l} = 0
    endfor
endfor
// Iterate until the convergence criteria is met
for i = 1 to MAXITS do
    rnorm = 0
    // Update values of u_{j,l} using red-black ordering
    jsw = 1
    for ipass = 1 to 2 do
        lsw = jsw
        for j = 2 to N - 1 do
            for l = lsw + 1 to N - 1 by 2 do
               r_{j,l} = u_{j+1,l} + u_{j-1,l} + u_{j,l+1} + u_{j,l-1} - 4u_{j,l}
               rnorm = rnorm + fabs(r_{j,l})
               u_{j,l} = u_{j,l} - ω * r_{j,l}/ - 4
            endfor
            lsw = 3 - lsw
        endfor
        jsw = 3 - jsw
        // adjust over-relaxation parameter ω
        ω = adjust(ω)
    endfor
    if rnorm < ε then return
endfor
error ("Iteration number limit exceeded")
```

Fig. 3. Algorithm for solving Dirichlet problem using SOR

After discretization we get a discrete problem of type 2 with the coefficients $a_{j,l} = b_{j,l} = c_{j,l} = d_{j,l} = 1, e_{j,l} = -4$ and $f_{j,l} = 0$.

This problem can be solved iteratively using one of the relaxation methods. We shall consider solving it using Successive Over-relaxation (SOR) with Chebyshev acceleration [8]. The algorithm is shown in Figure 3.

At each iteration the new values $u_{j,l}^{i+1}$ are computed from the old values $u_{j,l}^i$,$u_{j+1,l}^i$,$u_{j-1,l}^i$, $u_{j,l+1}^i$ and $u_{j,l-1}^i$. The values $u_{j,l}$ are updated in so called black-red order, when, first all $u_{j,l}$ s.t. $j + l$ is even ("black squares of the checkerboard") are processed, and then all $u_{j,l}$ s.t. $j + l$ is odd ("red squares") are processed.

The algorithm stops when the 1-norm of the residual r becomes sufficiently small: $\|r^i\|_1 \leq \varepsilon$.

4 Results

We performed experiments on several example codes to determine the effectiveness of this technique. We collected statistics based on uniprocessor runs to determine how often the technique may be useful, and we applied the technique by hand to two programs.

We ran an instrumented version of the SOR sample program on a uniprocessor machine to examine its convergence behavior. We assumed a data distribution that distributes columns of the `u` array over 16 processors. We modified the global reduction to perform each of the 16 reductions that would take place on local processors, then examined how useful the information would be in determining non-convergence locally.

We found that for the normal ordering, the sample code converged in 1141 iterations of the `while` loop. For the first 906 iterations, or 79.4%, all 16 processors were able to detect that the computation had not yet converged from purely local information. A majority of processors could determine this for the first 1026 iterations, or 89.3% of the total iterations.

For the red-black ordering, the sample code converged in 1028 iterations, and all processors could determine non-convergence locally for the first 902 iterations, or 87.7%. A majority of the processors could detect non-convergence for 937 iterations, or 91.1%.

So, for the greatest part of the computation, our optimization allows the program to avoid 90% of the global reductions (and the barriers associated with them). Furthermore, on the normal ordering, the optimization would allow us to use doacross-style parallelism for most of the program, where little parallelism was previously available.

We also examined the benchmark program `tomcatv` from the SPEC benchmarks. This program computes the infinity norm over two arrays, and exits when the norm falls below a value `eps`. The code does not converge given the test data, and runs for 100 iterations; thus, running it demonstrates the maximum potential gain from avoiding the reduction, and does not measure performance for the portion of computation closer to convergence.

#Processors	IP		User space	
	Optimized	Unoptimized	Optimized	Unoptimized
2	189	181	212	180
4	97	95	130	96
8	53	57	66	49
12	40	55	43	33
16	37	73	36	25

Table 1. Execution times in seconds for `tomcatv` (size 1025) on the SP2

We implemented a straightforward message-passing version of `tomcatv`, plus several latency-tolerating transformations, to produce a baseline version. We then applied our transformation to that program, and compared the two.

Experiments were performed on a 16-processor IBM SP2, using the `MPIF` library for communications. We first examined which processors could determine non-convergence in isolation. For a problem size of `n=257`, using any number of processors up to 16, all processors can determine convergence in every iteration. For a problem size of `n=513`, running on 16 processors, the last processor cannot detect non-convergence in iterations 36 - 100, and must communicate.

Table 1 shows results of running `tomcatv` with problem size 1025 under two different communications libraries on the SP2. Using the IP protocol for communication, which has a higher latency and overhead, the transformation improved performance, particularly on larger numbers of processors. Under the faster user space (US) protocol, however, the transformed code ran slower than the original. Contrary to our expectations, using looser synchronization to permit overlap resulted in worse performance than using barriers: when we inserted a barrier in the transformed code, it removed any opportunity for overlap, but improved performance. We speculated on the cause of this behavior, but were not able to conclusively determine the cause. Since the transformation is predicated on the assumption that removing barriers increases performance, it should not be used on systems where that assumption does not hold.

We also implemented a message-passing parallel version of SOR. The results are displayed in Figure 4. The best speedup on 15 processors (including a server process) was 5.8. The last portion of the computation, where reduction communication is required, is at least partially sequentialized. To examine the effect of that portion of the computation, we ran versions of the program that terminate after 1000 iterations, before any processor has to request assistance from the server (labeled as 1k on the graph). That version showed speedups to 7.7 on 15 processors. Since the version which computes until convergence comes reasonably close to the performance of the version which performs no reductions, improvements in performance are likely to come in areas unrelated to the reduction and convergence computation.

We also examined some more complicated applications to see what would be required to apply the transformation. We looked at the serial version of `bt`, one of the sample applications from the NAS Parallel Benchmarks. In this code,

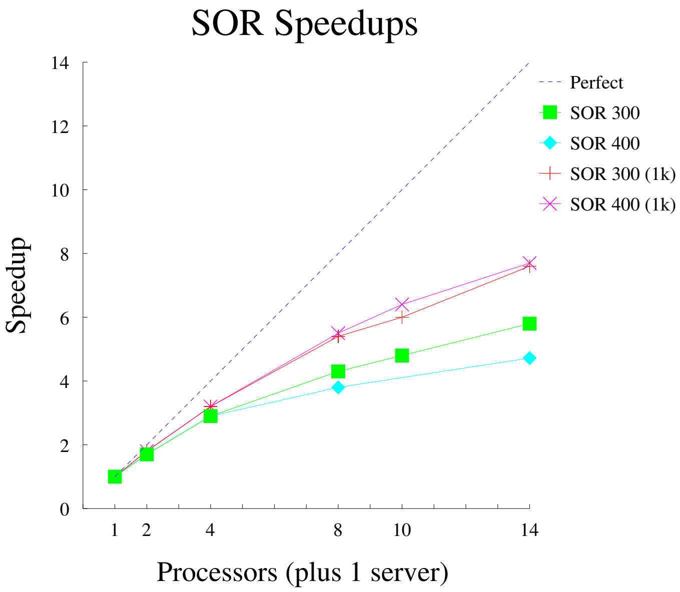

Fig. 4. Speedups for SOR on 16-processor SP2

in the `badi` subroutine, a vector of five norms is computed, and convergence is based upon all five meeting the convergence criteria. In addition, the computation of the norms takes place across a number of procedures. In order to be effective as an automatic transformation on this code, it would probably be necessary to both recognize more complicated convergence functions, and use interprocedural analysis to determine both possibility and profitability of applying the optimization; alternatively, users could supply directives to request it.

5 Conclusion

We have presented a method for reducing global reductions and increasing opportunities for doacross-style parallelism in certain kinds of iterative programs. The situation we have described, a monotone convergence test, arises frequently in real numerical applications. The techniques we have described allow us to avoid the cost of a barrier synchronization for most of the computation, until global information is necessary to determine if the computation has converged.

Our technique also allows us to provide efficient doacross/pipelined parallelism when the body of a while loop contains cross-processor dependencies. We believe the technique we propose is more practical than speculative execution [1,4,5,2,7,3].

In a language like HPF, the transformation we describe has to be performed by the compiler; there is no way for the user to express a reduction over local data and make a decision based on that.

In the experiments we performed, for most of the computation, local data alone was sufficient to determine that the algorithm had not yet converged. We also found that the technique can improve performance for both the case with dependences and without. On systems where removing barriers may decrease performance, it should not be applied.

In computations with convergence tests, other transformations are possible (for example, checking for convergence only every ten iterations). While these transformations may be useful, they can change the results of some computations and we believe they should not be done without the users involvement and concurrence.

References

1. J.-F. Collard. Space-time transformation of while-loops using speculative execution. In *Proc. of the 1994 Scalable High Performance Computing Conf.*, pages 429–436, Knoxville, TN, May 1994. IEEE.
2. J.-F. Collard. Automatic parallelization of while-loops using speculative execution. *Int. J. of Parallel Programming*, 23(2):191–219, April 1995.
3. M. Griebl and J.-F. Collard. Generation of synchronous code for automatic parallelization of while loops. In N.N., editor, *EuroPar 95*, Lecture Notes in Computer Science. Springer-Verlag, 1995.
4. M. Griebl and C. Lengauer. On scanning space-time mapped **while** loops. In B. Buchberger and J. Volkert, editors, *Parallel Processing: CONPAR 94 – VAPP VI*, Lecture Notes in Computer Science 854, pages 677–688. Springer-Verlag, 1994.
5. M. Griebl and C. Lengauer. On the space-time mapping of WHILE-loops. *Parallel Processing Letters*, 4(3):221–232, September 1994.
6. David Kincaid and Ward Cheney. *Numerical Analysis*. Brooks/Cole Publishing Company, 1991.
7. C. Lengauer and M. Griebl. On the parallelization of loop nests containing while loops. In N. N. Mirenkov, Q.-P. Gu, S. Peng, and S. Sedukhin, editors, *Proc. 1st Aizu Int. Symp. on Parallel Algorithm/Architecture Synthesis (pAs'95)*, pages 10–18. IEEE Computer Society Press, 1995.
8. William H. Press, Saul A. Teukolsky, William T. Vetterling, and Brian P. Flannery. *Numerical Recipes in C: The Art of Scientific Computing*. Cambridge University Press, second edition, 1992.